REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Final: 09 Jan 1991 to 01 Mar 1993

4. TITLE AND SUBTITLE
Tartan Inc., Tartan Ada Sun/Sun, Version 4.0, Sun3/60 , SunOS version 4.0.3 (Host & Target), 901211I1.11118

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, Federal Republic of Germany

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION REPORT NUMBER
IABG-VSR 077

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Tartan In., Tartam Ada Sun/Sun Version 4.0 Sun3/60, Ottobrunn, Germany, SunOS Version 4.0.3 (Host & Target), ACVC 1.11.

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on December 11, 1990.

Compiler Name and Version: Tartan Ada Sun/Sun version 4.0

Host Computer System: Sun3/60 SunOS version 4.0.3

Target Computer System: Sun3/60 SunOS version 4.0.3

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901211I1.11118 is awarded to Tartan Inc. This certificate expires on 01 March, 1993.

This report has been reviewed and is approved.

Michael Tonndorf

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

for

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

John P. Solomond

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

AVF Control Number: IABG-VSR 077
9 January, 1991

== based on TEMPLATE Version 90-08-15 ==

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901211I1.11118
Tartan Inc.
Tartan Ada Sun/Sun version 4.0
Sun3/60 SunOS version 4.0.3
Host & Target

Prepared By:
IABG, ABT. ITE

DECLARATION OF CONFORMANCE

Customer: Tartan, Inc.

Certificate Awardee: Tartan, Inc.

Ada Validation Facility: IABG

ACVC Version: 1.11

Ada Implementation:

Ada Compiler Name and Version: Tartan Ada Sun/Sun Version 4.0

Host Compiler System: Sun 3/60 SunOS Version 4.0.3

Target Computer System: Sun 3/60 SunOS Version 4.0.3

Declaration:

[I/we] the undersigned, declare that [I/we] have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

Lee Ehlert

Customer Signature

Date: 12/14/90

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in Section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see Section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process or service of all requirements specified.

Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is November 21, 1990.

E28005C	B28006C	C34006D	C35702A	B41308B	C43004A
C45114A	C45346A	C45612B	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	B85001L	C83026A	C83041A	C97116A	C98003B
BA2011A	CB7001A	CB7001B	CB7004A	CC1223A	BC1226A
CC1226B	BC3009B	AD1B08A	BD1B02B	BD1B06A	BD2A02A
CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A
CD2B15C	BD3006A	BD4008A	CD4022A	CD4022D	CD4024B
CD4024C	CD4024D	CD4031A	CD4051D	CD5111A	CD7004C
ED7005D	CD7005E	AD7006A	CD7006E	AD7201A	AD7201E
CD7204B	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, there is no such type.

C45536A, C46013B, C46031B, C46033B, and C46034B contain 'SMALL representation clauses which are not powers of two or ten.

C45624A and C45624B are not applicable as MACHINE_OVERFLOWS is TRUE for floating-point types.

B86001Y checks for a predefined fixed-point type other than DURATION.

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C instantiate generic units before their bodies are compiled; this implementation creates a dependence on generic units as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (see 2.3.)

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten type'small; this implementation does not support decimal smalls. (see 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than what the length clause specified, as allowed by AI-00558.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

AE2101C and EE2201D..E (2 tests) use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

The tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3111B and CE3115A associate multiple internal text files with the same external file and attempt to read from one file what was written to the other, which is assumed to be immediately available; this implementation buffers output. (see 2.3.)

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see Section 1.3) were required for 109 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24007A	B24009A	B25002B	B32201A	B33204A
B33205A	B35701A	B36171A	B36201A	B37101A	B37102A
B37201A	B37202A	B37203A	B37302A	B38003A	B38003B
B38008A	B38008B	B38009A	B38009B	B38103A	B38103B
B38103C	B38103D	B38103E	B43202C	B44002A	B48002A
B48002B	B48002D	B48002E	B48002G	B48003E	B49003A
B49005A	B49006A	B49006B	B49007A	B49007B	B49009A
B4A010C	B54A20A	B54A25A	B58002A	B58002B	B59001A
B59001C	B59001I	B62006C	B67001A	B67001B	B67001C
B67001D	B74103E	B74104A	B74307B	B83E01A	B83E01B
B85007C	B85008G	B85008H	B91004A	B91005A	B95003A
B95007B	B95031A	B95074E	BC1002A	BC1109A	BC1109C
BC1206A	BC2001E	BC3005B	BD2A06A	BD2B03A	BD2D03A
BD4003A	BD4006A				

E28002B was graded inapplicable by Evaluation and Test Modification as directed by the AVO. This test checks that pragmas may have unresolvable arguments, and it includes a check that pragma LIST has the required effect; but for this implementation, pragma LIST has no effect if the compilation results in errors or warnings, which is the case when the test is processed without modification. This test was also processed with the pragmas at lines 46, 58, 70 and 71 commented out so that pragma LIST had effect.

Tests C45524A..K (11 tests) were graded passed by Test Modification as directed by the AVO. These tests expect that a repeated division will result in zero; but the standard only requires that the result lie in the smallest safe interval. Thus, the tests were modified to check that the result was within the smallest safe interval by adding the following code after line 141; the modified tests were passed:

```
ELSIF VAL <= F'SAFE_SMALL THEN COMMENT ("UNDERFLOW SEEMS GRADUAL");
```

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package report's body, and thus the packages' calls to function Report.Ident_Int at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

B83E01B was graded passed by Evaluation Modification as directed by the AVO. This test checks that a generic subprogram's formal parameter names (i.e. both generic and subprogram formal parameter names) must be distinct; the duplicated names within the generic declarations are marked as errors, whereas their recurrences in the subprogram bodies are marked as "optional" errors--except for the case at line 122, which is marked as an error. This implementation does not additionally flag the errors in the bodies and thus the expected error at line 122 is not flagged. The AVO ruled that the implementation's behavior was acceptable and that the test need not be split (such a split would simply duplicate the case in B83E01A at line 15).

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C were graded inapplicable by Evaluation Modification as directed by the AVO. These tests instantiate generic units before those units' bodies are compiled; this implementation creates dependences as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete, and the objectives of these tests cannot be met.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 & AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by compiling the seperate files in the following order (to allow re-compilation of obsolete units), and all intended errors were then detected by the compiler:

BC3204C: C0, C1, C2, C3M, C4, C5, C6, C3M

BC3205D: D0, D2, D1M

BC3204D and BC3205C were graded passed by Test Modification as directed by the AVO. These tests are similar to BC3204C and BC3205D above, except that all compilation units are contained in a single compilation. For these two tests, a copy of the main procedure (which later units make obsolete) was appended to the tests; all expected errors were then detected.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-ten value as small for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal smalls may be omitted.

AD9001B and AD9004A were graded passed by Processing Modification as directed by the AVO. These tests check that various subprograms may be interfaced to external routines (and hence have no Ada bodies). This implementation requires that a file specification exists for the foreign subprogram bodies. The following command was issued to the Librarian to inform it that the foreign bodies will be supplied at link time (as the bodies are not actually needed by the program, this command alone is sufficient:

```
adalib> interface -sys -L=library AD9004A
```

CE3111B and CE3115A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests assume that output from one internal file is unbuffered and may be immediately read by another file that shares the same external file. This implementation raises END_ERROR on the attempts to read at lines 87 & 101, respectively.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Mr Ron Duursma
Director of Ada Products
Tartan Inc.
300, Oxford Drive,
Monroeville, PA 15146,
USA.
Tel. (412) 856-3600

For a point of contact for sales information about this Ada implementation system, see:

Mr Bill Geese
Director of Sales
Tartan Inc.
300, Oxford Drive,
Monroeville, PA 15146,
USA.
Tel. (412) 856-3600

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3792	
b) Total Number of Withdrawn Tests	83	
c) Processed Inapplicable Tests	94	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	295	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in Section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in Section 2.1 had been withdrawn because of test errors. The AVF determined that 295 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in Section 2.3 were also processed.

A 1/4" Data Cartridge containing the customized test suite (see Section 1.3) was taken on-site by the validation team for processing. The contents of the tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

Options used for compiling:

-f forces the compiler to accept an attempt to compile a unit imported from another library, which is normally prohibited.

-q quiet, stops output of all compiler phase names. Not documented in product version as it is the default setting. Option -v was the default setting for the validation run.

-c normally the compiler creates a registered copy of the user's source code in the library directory for proper operation of the remake and make subcommands to Adalib.

-La forces a compiler to produce a listing even if no errors were found.

No explicit linker Options were used.

Test output, compiler and linker listings, and job logs were captured on a 1/4" Data Cartridge and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$MAX_IN_LEN	240
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2147483646
$DEFAULT_MEM_SIZE	1_000_000
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	MC68000
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	16#24_004#
$ENTRY_ADDRESS1	16#24_008#
$ENTRY_ADDRESS2	16#24_00C#
$FIELD_LAST	20
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	THERE_IS_NO_SUCH_FLOAT_NAME
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	100_000_000.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E+38

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        1.0E+38

$HIGH_PRIORITY          200

$ILLEGAL_EXTERNAL_FILE_NAME1
                       ILLEGAL_EXTERNAL_FILE_NAME1----------------

$ILLEGAL_EXTERNAL_FILE_NAME2
                       ILLEGAL_EXTERNAL_FILE_NAME2----------------

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        "PRAGMA INCLUDE ("A28006D1.TST")"

$INCLUDE_PRAGMA2        "PRAGMA INCLUDE ("B28006F1.TST")"

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483647

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     NO_LANGUAGE

$LESS_THAN_DURATION     -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -100_000_000.0

$LINE_TERMINATOR        ASCII.LF

$LOW_PRIORITY           10

$MACHINE_CODE_STATEMENT
                        NULL;

$MACHINE_CODE_TYPE      NO_SUCH_TYPE

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                -2147483648
```

```
$NAME                   BYTE_INTEGER

$NAME_LIST              VAX,MIL_STD_1750A,MC68000,ND500

$NAME_SPECIFICATION1    DISK$AWC_2:[CROCKL.ACVC11.DEVELOP]X2120A.;1

$NAME_SPECIFICATION2    DISK$AWC_2:[CROCKL.ACVC11.DEVELOP]X2120B.;1

$NAME_SPECIFICATION3    DISK$AWC_2:[CROCKL.ACVC11.DEVELOP]X2120C.;1

$NEG_BASED_INT          8#777777777776#

$NEW_MEM_SIZE           500_000

$NEW_STOR_UNIT          8

$NEW_SYS_NAME           VAX

$PAGE_TERMINATOR        ASCII.FF

$RECORD_DEFINITION      NEW INTEGER;

$RECORD_NAME            NO_SUCH_MACHINE_CODE_TYPE

$TASK_SIZE              96

$TASK_STORAGE_SIZE      1024

$TICK                   0.01667

$VARIABLE_ADDRESS       16#24_000#

$VARIABLE_ADDRESS1      16#24_030#

$VARIABLE_ADDRESS2      16#24_034#

$YOUR_PRAGMA            NO_SUCH_PRAGMA
```

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

Compilation switches for Tartan Ada Sun-Sun.

-a	Generate an assembly code file. The assembly code file has an extension .s for a body or .ss for a specification (see Section 4.4).
-A	Generate an assembly code file with interleaved source code. The assembly code file has an extension .s for a body or .ss for a specification.
-c	Normally, the compiler creates a registered copy of the user's source code in the library directory for proper operation of the remake and make subcommands to adalib. This option suppresses the creation of this copy.
-d	When compiling a library unit, determine whether the unit is a refinement of its previous version and, if so, do not make dependent units obsolete. This check is not done by default.
-e=<integer>	Stop compilation and produce a listing after n errors are encountered, where n is in the range 0..255. The default value for n is 255. The -e qualifier cannot be negated.
-f	Forces the compiler to accept an attempt to compile a unit imported from another library, which is normally prohibited.
-g	Compile with debugging information for AdaScope.
-i	Cause compiler to omit data segments with the text of enumeration literals. This text is normally produced for exported enumeration types in order to support the text attributes ('IMAGE, 'VALUE and 'WIDTH). You should use -i only when you can guarantee that no unit that will import the enumeration type will use any of its text attributes. However, if you are compiling a unit with an enumeration type that is not visible to other compilation units, this option is not needed. The compiler can recognize when the text attributes are not used and will not generate the supporting strings.
-L=[project:]library	Select library and/or project for this compilation. This option takes effect after all commands from the .adalibrc file have been executed, thereby possibly overriding its effects.

-La — Generate a listing, even if no errors were found. The default is to generate a listing only if an error is found.

-Ln — Never generate a listing. The default is to generate a listing only if an error is found.

-Op=n — Control the level of optimization performed by the compiler, requested by n. The optimization levels available are:

- n = 0 — Minimum - Performs context determination, constant folding, algebraic manipulation, and short circuit analysis.
- n = 1 — Low - Performs level 0 optimizations plus common subexpression elimination and equivalence propagation within basic blocks. It also optimizes evaluation order.
- n = 2 — Best tradeoff for space/time - the default level. Performs level 1 optimizations plus flow analysis which is used for common subexpression elimination and equivalence propagation across basic blocks. It also performs invariant expression hoisting, dead code elimination, and assignment killing. Level 2 also performs lifetime analysis which is used to improve register allocation. It also performs inline expansion of subprogram calls indicated by Pragma INLINE, if possible.
- n = 3 — Time - Performs level 2 optimizations plus inline expansion of subprogram calls which the optimizer decides are profitable to expand (from an execution time perspective). Other optimizations which improve execution time at a cost to image size are performed
- n = 4 — Space - Performs those optimizations which usually produce the smallest code, often at the expense of speed. This optimization level may not always produce

the smallest code, however, another level may produce smaller code under certain conditions.

-r — For internal use only, this option is used by adalib when it invokes the compiler in (re)make mode.

-S[ACDEILORSZ] — Suppress the given set of checks:

A	ACCESS_CHECK
C	CONSTRAINT_CHECK
D	DISCRIMINANT_CHECK
E	ELABORATION_CHECK
I	INDEX_CHECK
L	LENGTH_CHECK
O	OVERFLOW_CHECK
R	RANGE_CHECK
S	STORAGE_CHECK
Z	"ZERO"DIVISION_CHECK

The -S option has the same effect as an equivalent pragma SUPPRESS applied to the source file. If the source program also contains a pragma SUPPRESS, then a given check is suppressed if either the pragma or the switch specifies it; that is, the effect of a pragma SUPPRESS cannot be negated with the command line option. See LRM 11.7 for further details. Supplying the -S option significantly decreases the size and execution time of the compiled code. Examples are:

-SOZ — Suppress OVERFLOW_CHECK and "ZERO"DIVISION_CHECK.

-S — Suppress all checks.

-SC — Suppress CONSTRAINT_ERROR, equivalent to -SADILR. (Note that -SC is upward compatible with Version 2.0)

-v — Print out compiler phase names. The compiler prints out a short description of each compilation phase in progress.

-w — Suppress warning messages.

-x — Include cross reference information for the source in the object file (see section 4.6).

In addition, the output from the compiler may be redirected using the SunOS redirection facility including '&' for stderr; for example

```
% tada tax_spec.ada >& tax_spec.txt
```

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

Linker switches for Sun hosted Tartan Ada compilers.

COMMAND QUALIFIERS

This section describes the command options available to a user who directly invokes the linker. The option names can be abbreviated to unique prefixes; the first letter is sufficent for all current option names. The option names are not case sensitive.

-CONTROL file	The specified file contains linker control commands. Only one such file may be specified, but it can include other files using the CONTROL command. Every invocation of the linker must specify a control file.
-OUTPUT file	The specified file is the name of the first output object file. The module name for this file will be null. Only one output file may be specified in this manner. Additional output files may be specified in the linker control file.
-ALLOCATIONS	Produce a link map showing the section allocations.
-UNUSEDSECTIONS	Produce a link map showing the unused sections.
-SYMBOLS	Produce a link map showing global and external symbols.
-RESOLVEMODULES	This causes the linker to not perform unused section elimination. Specifying this option will generally make your program larger, since unreferenced data within object files will not be eliminated. Refer to Sections RESOLVE_CMD and USE_PROCESSING for information on the way that unused section elimination works.
-MAP	Produce a link map containing all information except the unused section listings.

Note that several listing options are permitted. This is because link maps for real systems can become rather large, and writing them consumes a significant fraction of the total link time. Options specifying the contents of the link map can be combined, in which case the resulting map will contain all the information specified by any of the switches. The name of the file containing the link map is specified by the LIST command in the linker control file. If your control file does not specify a name and you request a listing, the listing will be written to the standard output stream.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. The package STANDARD is presented in this implementation's Appendix F on page 5-11.

Chapter 5
Appendix F to MIL-STD-1815A

This chapter contains the required Appendix F to the LRM which is *Military Standard, Ada Programming Language*, ANSI/MIL-STD-1815A (American National Standards Institute, Inc., February 17, 1983).

5.1. *PRAGMAS*

5.1.1. Predefined Pragmas

This section summarizes the effects of and restrictions on predefined pragmas.

- Access collections are not subject to automatic storage reclamation so pragma `CONTROLLED` has no effect. Space deallocated by means of `UNCHECKED_DEALLOCATION` will be reused by the allocation of new objects.
- Pragma `ELABORATE` is supported.
- Pragma `INLINE` is supported.
- Pragma `INTERFACE` is supported. It is assumed that the foreign code interfaced adheres to Tartan Ada calling conventions as well as Tartan Ada parameter passing mechanisms. Any `Language_Name` will be accepted, but ignored, and the default will be used.
- Pragma `LIST` is supported but has the intended effect only if the command line option `-La` was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma `MEMORY_SIZE` is accepted but no value other than that specified in Package `SYSTEM` (Section 5.3) is allowed.
- Pragma `OPTIMIZE` is supported except when at the outer level (that is, in a package specification or body).
- Pragma `PACK` is supported.
- Pragma `PAGE` is supported but has the intended effect only if the command line option `-La` was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.
- Pragma `PRIORITY` is supported.
- Pragma `STORAGE_UNIT` is accepted but no value other than that specified in Package `SYSTEM` (Section 5.3) is allowed.
- Pragma `SHARED` is not supported. No warning is issued if it is supplied.
- Pragma `SUPPRESS` is supported.
- Pragma `SYSTEM_NAME` is accepted but no value other than that specified in Package `SYSTEM` (Section 5.3) is allowed.

5.1.2. Implementation-Defined Pragmas

Implementation-defined pragmas provided by Tartan are described in the following sections.

5.1.2.1. Pragma LINKAGE_NAME

The pragma LINKAGE_NAME associates an Ada entity with a string that is meaningful externally; e.g., to a linkage editor. It takes the form

pragma LINKAGE_NAME (*Ada-simple-name, string-constant*)

The *Ada-simple-name* must be the name of an Ada entity declared in a package specification. This entity must be one that has a runtime representation; e.g., a subprogram, exception or object. It may not be a named number or string constant. The pragma must appear after the declaration of the entity in the same package specification.

The effect of the pragma is to cause the *string-constant* to be used in the generated assembly code as an external name for the associated Ada entity. It is the responsibility of the user to guarantee that this string constant is meaningful to the linkage editor and that no illegal linkname clashes arise.

This pragma has no effect when applied to a library subprogram or to a *renames* declaration; in the latter case, no warning message is given.

When determining the maximum allowable length for the external linkage name, keep in mind that the compiler will generate names for elaboration flags simply by appending the suffix #GOTO. Therefore, the external linkage name has 5 fewer significant characters than the lower limit of other tools that need to process the name (e.g., 40 in the case of the Tartan Linker).

5.1.2.2. Pragma FOREIGN_BODY

In addition to Pragma INTERFACE, Tartan Ada supplies Pragma FOREIGN_BODY as a way to access subprograms in other languages.

Unlike Pragma INTERFACE, Pragma FOREIGN_BODY allows access to objects and exceptions (in addition to subprograms) to and from other languages.

Some restrictions on Pragma FOREIGN_BODY that are not applicable to Pragma INTERFACE are:

- Pragma FOREIGN_BODY must appear in a non-generic library package.
- All objects, exceptions and subprograms in such a package must be supplied by a foreign object module.
- Types may not be declared in such a package.

Use of the pragma FOREIGN_BODY dictates that all subprograms, exceptions and objects in the package are provided by means of a foreign object module. In order to successfully link a program including a foreign body, the object module for that body must be provided to the library using the `adalib foreign` command described in sections 3.3.3 and 9.5.5. The pragma is of the form:

pragma FOREIGN_BODY (*Language_name* [, *elaboration_routine_name*])

The parameter *Language_name* is a string intended to allow the compiler to identify the calling convention used by the foreign module (but this functionality is not yet in operation). Currently, the programmer must ensure that the calling convention and data representation of the foreign body procedures are compatible with those used by the Tartan Ada compiler. Subprograms called by tasks should be reentrant.

The optional *elaboration_routine_name* string argument is a linkage name identifying a routine to initialize the package. The routine specified as the *elaboration_routine_name*, which will be called for the elaboration of this package body, must be a global routine in the object module provided by the user.

A specification that uses this pragma may contain only subprogram declarations, object decarations that use an unconstrained type mark, and number declarations. Pragmas may also appear in the package. The type mark for an object cannot be a task type, and the object declaration must not have an initial value expression. The pragma must be given prior to any declarations within the package specification. If the pragma is not located before the first declaration, or any restriction on the declarations is violated, the pragma is ignored and a warning is generated.

The foreign body is entirely responsible for initializing objects declared in a package utilizing pragma FOREIGN_BODY. In particular, the user should be aware that the implicit initializations described in LRM 3.2.1 are not done by the compiler. (These implicit initializations are associated with objects of access types, certain record types and composite types containing components of the preceding kinds of types.)

Pragma LINKAGE_NAME should be used for all declarations in the package, including any declarations in a nested package specification to be sure that there are no conflicting link names. If pragma LINKAGE_NAME is not used, the cross-reference qualifier, -x, (see Section 4.2) should be used when invoking the compiler and the resulting cross-reference table of linknames inspected to identify the linknames assigned by the compiler and determine that there are no conflicting linknames (see also Section 4.6). In the following example, we want to call a function plmn which computes polynomials and is written in C.

```
package MATH_FUNCS is
  pragma FOREIGN_BODY ("C");
  function POLYNOMIAL  (X:INTEGER) return INTEGER;
        --Ada spec matching the C routine
  pragma LINKAGE_NAME (POLYNOMIAL, "plmn");
        --Force compiler to use name "plmn" when referring to this
  -- function
end MATH_FUNCS;

with MATH_FUNCS; use MATH_FUNCS
procedure MAIN is
  X:INTEGER := POLYNOMIAL(10);
        -- Will generate a call to "plmn"
  begin ...
end MAIN;
```

To compile, link and run the above program, you do the following steps:

1. Compile MATH_FUNCS
2. Compile MAIN
3. Obtain an object module (e.g. math.tof) containing the compiled code for plmn, converted to Tartan Object File Format (TOFF) using the aout_to_toff utility (See *Object File Utilities*, Chapter 4).
4. Issue the command

   ```
   adalib foreign math_funcs math.tof
   ```

5. Issue the command

   ```
   adalib link main
   ```

Without Step 4, an attempt to link will produce an error message informing you of a missing package body for MATH_FUNCS.

Using an Ada body from another Ada program library. The user may compile a body written in Ada for a specification into the library, regardless of the language specified in the pragma contained in the specification. This capability is useful for rapid prototyping, where an Ada package may serve to provide a simulated response for the functionality that a foreign body may eventually produce. It also allows the user to replace a foreign body with an Ada body without recompiling the specification.

The user can either compile an Ada body into the library, or use the command adalib foreign (see Sections 3.3.3 and 9.5.5) to use an Ada body from another library. The Ada body from another library must have been compiled under an identical specification. The pragma LINKAGE_NAME must have been applied to all entities declared in the specification. The only way to specify the linkname for the elaboration routine of an Ada body is with the pragma FOREIGN_BODY.

5.2. *IMPLEMENTATION-DEPENDENT ATTRIBUTES*

No implementation-dependent attributes are currently supported.

5.3. *SPECIFICATION OF THE PACKAGE* SYSTEM

The parameter values specified for the SUN in package system [LRM 13.7.1 and Appendix C] are:

```
package SYSTEM is
   type ADDRESS is new INTEGER;
   type NAME is (MC68000);
   SYSTEM_NAME   : constant NAME := MC68000;
   STORAGE_UNIT  : constant := 8;
   MEMORY_SIZE   : constant := 1_000_000;
   MAX_INT       : constant := 2_147_483_647;
   MIN_INT       : constant := -MAX_INT - 1;
   MAX_DIGITS    : constant := 15;

   MAX_MANTISSA : constant := 31;
   FINE_DELTA   : constant := 2#1.0#e-31;
   TICK         : constant := 0.01667;
   subtype PRIORITY is INTEGER range 10 .. 200;
   DEFAULT_PRIORITY : constant PRIORITY := PRIORITY'FIRST;
   RUNTIME_ERROR    : exception;
end SYSTEM;
```

5.4. *RESTRICTIONS ON REPRESENTATION CLAUSES*

The following sections explain the basic restrictions for representation specifications followed by additional restrictions applying to specific kinds of clauses.

5.4.1. *Basic Restriction*

The basic restriction on representation specifications [LRM 13.1] is that they may be given only for types declared in terms of a type definition, excluding a `generic_type_definition` (LRM 12.1) and a `private_type_definition` (LRM 7.4). Any representation clause in violation of these rules is not obeyed by the compiler; an error message is issued.

Further restrictions are explained in the following sections. Any representation clauses violating those restrictions cause compilation to stop and a diagnostic message to be issued.

5.4.2. *Length Clauses*

Length clauses [LRM 13.2] are, in general, supported. For details, refer to the following sections.

5.4.2.1. *Size Specifications for Types*

The rules and restrictions for size specifications applied to types of various classes are described below.

The following principle rules apply:

1. The size is specified in bits and must be given by a static expression.
2. The specified size is taken as a mandate to store objects of the type in the given size wherever feasible. No attempt is made to store values of the type in a smaller size, even if possible. The following rules apply with regard to feasibility:

   - An object that is not a component of a composite object is allocated with a size and alignment that is referable on the target machine; that is, no attempt is made to create objects of non-referable size on the stack. If such stack compression is desired, it can be achieved by the user by combining multiple stack variables in a composite object; for example

```
type My_Enum is (A,B);
for My_enum'size use 1;
V,W: My_enum; -- will occupy two storage
              -- units on the stack
              -- (if allocated at all)
type rec is record
   V,W: My_enum;
end record;
pragma Pack(rec);
O: rec;       -- will occupy one storage unit
```

- A formal parameter of the type is sized according to calling conventions rather than size specifications of the type. Appropriate size conversions upon parameter passing take place automatically and are transparent to the user.

- Adjacent bits to an object that is a component of a composite object, but whose size is non-referable, may be affected by assignments to the object, unless these bits are occupied by other components of the composite object; that is, whenever possible, a component of non-referable size is made referable.

In all cases, the compiler generates correct code for all operations on objects of the type, even if they are stored with differing representational sizes in different contexts.

Note: A size specification cannot be used to force a certain size in value operations of the type; for example

```
type my_int is range 0..65535;
for my_int'size use 16; -- o.k.
A,B: my_int;
   ...A + B... -- this operation will generally be
               -- executed on 32-bit values
```

3. A size specification for a type specifies the size for objects of this type and of all its subtypes. For components of composite types, whose subtype would allow a shorter representation of the component, no attempt is made to take advantage of such shorter representations. In contrast, for types without a length clause, such components may be represented in a lesser number of bits than the number of bits required to represent all values of the type. Thus, in the example

```
type MY_INT is range 0..2**15-1;
for MY_INT'SIZE use 16; -- (1)
subtype SMALL_MY_INT is MY_INT range 0..255;
type R is record
            ...
           X: SMALL_MY_INT;
            ...
         end record;
```

the component R.X will occupy 16 bits. In the absence of the length clause at (1), R.X may be represented in 8 bits.

Size specifications for access types must coincide with the default size chosen by the complier for the type.

Size specifications are not supported for floating-point types or task types.

No useful effect can be achieved by using size specifications for these types.

5.4.2.2. Size Specification for Scalar Types

The specified size must accommodate all possible values of the type including the value 0 (even if 0 is not in the range of the values of the type). For numeric types with negative values the number of bits must account for the sign bit. No skewing of the representation is attempted. Thus

```
type my_int is range 100..101;
```

requires at least 7 bits, although it has only two values, while

```
type my_int is range -101..-100;
```

requires 8 bits to account for the sign bit.

A size specification for a real type does not affect the accuracy of operations on the type. Such influence should be exerted via the `accuracy_definition` of the type (LRM 3.5.7, 3.5.9).

A size specification for a scalar type may not specify a size larger than the largest operation size supported by the target architecture for the respective class of values of the type.

5.4.2.3. Size Specification for Array Types

A size specification for an array type must be large enough to accommodate all components of the array under the densest packing strategy. Any alignment constraints on the component type (see Section 5.4.7) must be met.

The size of the component type cannot be influenced by a length clause for an array. Within the limits of representing all possible values of the component subtype (but not necessarily of its type), the representation of components may, however, be reduced to the minimum number of bits, unless the component type carries a size specification.

If there is a size specification for the component type, but not for the array type, the component size is rounded up to a referable size, unless pragma PACK is given. This applies even to boolean types or other types that require only a single bit for the representation of all values.

5.4.2.4. Size Specification for Record Types

A size specification for a record type does not influence the default type mapping of a record type. The size must be at least as large as the number of bits determined by type mapping. Influence over packing of components can be exerted by means of (partial) record representation clauses or by Pragma PACK.

Neither the size of component types, nor the representation of component subtypes can be influenced by a length clause for a record.

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record or contain relative offsets of components within a record layout for record components of dynamic size. These implementation-dependent components cannot be named or sized by the user.

A size specification cannot be applied to a record type with components of dynamically determined size.

Note: Size specifications for records can be used only to widen the representation accomplished by padding at the beginning or end of the record. Any narrowing of the representation over default type mapping must be accomplished by representation clauses or pragma PACK.

5.4.2.5. Specification of Collection Sizes

The specification of a collection size causes the collection to be allocated with the specified size. It is expressed in storage units and need not be static; refer to package SYSTEM for the meaning of storage units.

Any attempt to allocate more objects than the collection can hold causes a STORAGE_ERROR exception to be raised. Dynamically sized records or arrays may carry hidden administrative storage requirements that must be accounted for as part of the collection size. Moreover, alignment constraints on the type of the allocated objects may make it impossible to use all memory locations of the allocated collection. No matter what the requested object size, the allocator must allocate a minimum of 2 words per object. This lower limit is necessary for administrative overhead in the allocator. For example, a request of 5 words results in an allocation of 5 words; a request of 1 word results in an allocation of 2 words.

In the absence of a specification of a collection size, the collection is extended automatically if more objects are allocated than possible in the collection originally allocated with the compiler-established default size. In this case, STORAGE_ERROR is raised only when the available target memory is exhausted. If a collection size of zero is specified, no access collection is allocated.

5.4.2.6. Specification of Task Activation Size

The specification of a task activation size causes the task activation to be allocated with the specified size. It is expressed in storage units; refer to package SYSTEM for the meaning of storage units.

Any attempt to exceed the activation size during execution causes a STORAGE_ERROR exception to be raised. Unlike collections, there is no extension of task activations.

5.4.2.7. Specification of 'SMALL

Only powers of 2 are allowed for 'SMALL.

The length of the representation may be affected by this specification. If a size specification is also given for the type, the size specification takes precedence; the specification of 'SMALL must then be accommodatable within the specified size.

5.4.3. Enumeration Representation Clauses

For enumeration representation clauses [LRM 13.3], the following restrictions apply:

- The internal codes specified for the literals of the enumeration type may be any integer value between INTEGER'FIRST and INTEGER'LAST. It is strongly advised to not provide a representation clause that merely duplicates the default mapping of enumeration types, which assigns consecutive numbers in ascending order starting with 0, since unnecessary runtime cost is incurred by such duplication. It should be noted that the use of attributes on enumeration types with user-specified encodings is costly at run time.
- Array types, whose index type is an enumeration type with non-contiguous value encodings, consist of a contiguous sequence of components. Indexing into the array involves a runtime translation of the index value into the corresponding position value of the enumeration type.

5.4.4. Record Representation Clauses

The alignment clause of record representation clauses [LRM 13.4] is observed.

Static objects may be aligned at powers of 2 up to a page boundary. The specified alignment becomes the minimum alignment of the record type, unless the minimum alignment of the record forced by the component allocation and the minimum alignment requirements of the components is already more stringent than the specified alignment.

The component clauses of record representation clauses are allowed only for components and discriminants of statically determinable size. Not all components need to be present. Component clauses for components of variant parts are allowed only if the size of the record type is statically determinable for every variant.

The size specified for each component must be sufficient to allocate all possible values of the component subtype (but not necessarily the component type). The location specified must be compatible with any alignment constraints of the component type; an alignment constraint on a component type may cause an implicit alignment constraint on the record type itself.

If some, but not all, discriminants and components of a record type are described by a component clause, then the discriminants and components without component clauses are allocated after those with component clauses; no attempt is made to utilize gaps left by the user-provided allocation.

5.4.5. Address clauses

Address clauses [LRM 13.5] are supported with the following restrictions:

- When applied to an object, an address clause becomes a linker directive to allocate the object at the given address. For any object not declared immediately within a top-level library package, the address clause is meaningless. Address clauses applied to local packages are not supported by Tartan Ada. Address clauses applied to library packages are prohibited by the syntax; therefore, an address clause can be applied to a package only if it is a body stub.

- Address clauses applied to subprograms and tasks are implemented according to the LRM rules. When applied to an entry, the specified value identifies an interrupt in a manner explained in section 5.6.
- A specified address must be an Ada static expression.

5.4.6. Pragma PACK

Pragma PACK [LRM 13.1] is supported. For details, refer to the following sections.

5.4.6.1. Pragma PACK *for Arrays*

If pragma PACK is applied to an array, the densest possible representation is chosen. For details of packing, refer to the explanation of size specifications for arrays (Section 5.4.2.3).

If, in addition, a length clause is applied to

1. The array type, the pragma has no effect, since such a length clause already uniquely determines the array packing method.
2. The component type, the array is packed densely, observing the component's length clause. Note that the component length clause may have the effect of preventing the compiler from packing as densely as would be the default if pragma PACK is applied where there was no length clause given for the component type.

5.4.6.2. The Predefined Type String

Package STANDARD applies Pragma PACK to the type `string`.

However, when applied to character arrays, this pragma cannot be used to achieve denser packing than is the default for the target: 1 character per 8-bit word.

5.4.6.3. Pragma PACK *for Records*

If pragma PACK is applied to a record, the densest possible representation is chosen that is compatible with the sizes and alignment constraints of the individual component types. Pragma PACK has an effect only if the sizes of some component types are specified explicitly by size specifications and are of non-referable nature. In the absence of pragma PACK, such components generally consume a referable amount of space.

It should be noted that the default type mapping for records maps components of boolean or other types that require only a single bit to a single bit in the record layout, if there are multiple such components in a record. Otherwise, it allocates a referable amount of storage to the component.

If pragma PACK is applied to a record for which a record representation clause has been given detailing the allocation of some but not all components, the pragma PACK affects only the components whose allocation has not been detailed. Moreover, the strategy of not utilizing gaps between explicitly allocated components still applies.

5.4.7. Minimal Alignment for Types

Certain alignment properties of values of certain types are enforced by the type mapping rules. Any representation specification that cannot be satisfied within these constraints is not obeyed by the compiler and is appropriately diagnosed.

Alignment constraints are caused by properties of the target architecture, most notably by the capability to extract non-aligned component values from composite values in a reasonably efficient manner. Typically, restrictions exist that make extraction of values that cross certain address boundaries very expensive, especially in contexts involving array indexing. Permitting data layouts that require such complicated extractions may impact code quality on a broader scale than merely in the local context of such extractions.

Instead of describing the precise algorithm of establishing the minimal alignment of types, we provide the general rule that is being enforced by the alignment rules:

- No object of scalar type including components or subcomponents of a composite type, may span a target-dependent address boundary that would mandate an extraction of the object's value to be performed by two or more extractions.

5.5. *IMPLEMENTATION-GENERATED COMPONENTS IN RECORDS*

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record. These components cannot be named by the user.

5.6. *INTERPRETATION OF EXPRESSIONS APPEARING IN ADDRESS CLAUSES*

Section 13.5.1 of the Ada Language Reference Manual describes a syntax for associating interrupts with task entries. Tartan Ada implements the address clause

for `TOENTRY` **use at** `intID;`

by associating the interrupt specified by `intID` with the `toentry` entry of the task containing this address clause. The interpretation of `intID` is both machine and compiler dependent.

Task entries can be bound to SunOS Signals via the **for ... use at** clause. See `man signal` for an explanation of the SunOS signal mechanism.

Using the address clause

for ... use at `SIGNAL_NUMBER`

signals such as Control C can be bound to Ada handlers which allow for continuation of program execution, or orderly termination at the user's discretion. An example of trapping Control C (`SIGINT` or 2) to an Ada handler follows:

```
with SYSTEM; use SYSTEM;
with text_io; use text_io;

procedure int2 is
    task taskA is
        entry P;
        for P use at 2;
    end taskA;
    task body taskA is
    begin
        Put_Line("Package Interrupt Test");
        loop
            accept P do
                Put_Line("Ada found SIGINT (Control C)");
            end;
        end loop;
    end taskA;

begin
    for i in 1..30 loop
        delay 1.0;
    end loop;
    abort taskA;
end int2;
```

All currently recognized SunOS signals may be bound to task entries, with the signal numbers ranging from 1 to 31. In cases where the Ada runtimes assume a default operation for a given signal, the user's Ada trap may prevent normal execution of Ada programs.

The most critical example of this situation is in the case `SIGALRM`, that is, the SunOS Alarm Clock signal. The Ada runtimes use this signal to allow delayed tasks to wake up on schedule. If the user overwrites this handler with a replacement `SIGALRM` handler, *no tasks may be delayed.* Undesirable, and, depending upon the user's program, unknown execution will result. `SIGALRM` may be sucessfully mapped to a user's Ada handler in

cases where no tasks are delayed and the user would like to "wake up" the task entry that serves as the SIGALRM handler via the SIGALRM mechanism.

Other examples of default runtime functions for certain signals all involve the mapping of signals into Ada exceptions. The user has the option, for these signals, to place a new signal handler in the Ada exception, or to override the raise of the Ada exception, by writing a replacement handler for the signal. Currently, signals which raise Ada exceptions are SIGEMT, which raises NUMERIC_ERROR, and SIGFPE, which raises RUNTIME_ERROR.

Currently, for all signals other than those mentioned above, there is no default handler, and a raise of the signal by the operating system will result in the behavior as described in the SunOS manual for that signal. In most cases, this is an abrupt termination of the program, possibly resulting in a core image dump.

5.7. *RESTRICTIONS ON UNCHECKED CONVERSIONS*

Tartan supports UNCHECKED_CONVERSION with a restriction that requires the sizes of both source and target types to be known at compile time. The sizes need not be the same. If the value in the source is wider than that in the target, the source value will be truncated. If narrower, it will be zero-extended. Calls on instantiations of UNCHECKED_CONVERSION are made inline automatically.

5.8. *IMPLEMENTATION-DEPENDENT ASPECTS OF INPUT-OUTPUT PACKAGES*

Tartan Ada supports all predefined input/output packages [LRM Chapter 14] with the exception of LOW_LEVEL_IO.

SEQUENTIAL_IO and DIRECT_IO may not be instantiated on types whose representation size is greater than 32255 bytes. Any attempt to read or write values of such types raises USE_ERROR.

SEQUENTIAL_IO and DIRECT_IO may not be instantiated on unconstrained array types, nor on record record types with discriminants without default values.

An attempt to delete an external file while more than one internal file refers to this external file raises USE_ERROR.

When an external file is referenced by more than one internal file, an attempt to reset one of those internal files to OUT_FILE raises USE_ERROR.

An attempt to create a file with FILE_MODE IN_FILE raises USE_ERROR.

Since the implementation of the input-output packages uses buffers, output to one file cannot necessarily be read immediately from another file associated with the same external file.

The FORM parameter of file management subprograms is ignored.

An attempt to read a non-existent data record through the operations of SEQUENTIAL_IO or DIRECT_IO raises DATA_ERROR, except that END_ERROR is raised when reading beyond the end of file.

If a SunOS system call returns an error number that cannot be mapped onto a predefined Ada exception, the exception DEVICE_ERROR is raised.

5.9. *OTHER IMPLEMENTATION CHARACTERISTICS*

The following information is supplied in addition to that required by Appendix F to MIL-STD-1815A.

5.9.1. Definition of a Main Program

Any Ada library subprogram unit may be designated the main program for purposes of linking (using the adalib LINK command) provided that the subprogram has no parameters.

Tasks initiated in imported library units follow the same rules for termination as other tasks [described in LRM 9.4 (6-10)]. Specifically, these tasks are not terminated simply because the main program has terminated. Terminate alternatives in selective wait statements in library tasks are therefore strongly recommended.

5.9.2. Implementation of Generic Units

All instantiations of generic units, except the predefined generic UNCHECKED_CONVERSION and UNCHECKED_DEALLOCATION subprograms, are implemented by code duplications. No attempt at sharing code by multiple instantiations is made in this release of Tartan Ada.

Tartan Ada enforces the restriction that the body of a generic unit must be compiled before the unit can be instantiated. It does not impose the restriction that the specification and body of a generic unit must be provided as part of the same compilation. A recompilation of the body of a generic unit will casue any units that instantiated this generic unit to become obsolete.

5.9.3. Implementation-Defined Characteristics in Package STANDARD

The implementation-dependent characteristics for SUN in package STANDARD [Annex C] are:

```
package STANDARD is
...
type BYTE_INTEGER is range -128 .. 127;
type SHORT_INTEGER is range -32768 .. 32767;
type INTEGER is range -2_147_483_648 .. 2_147_483_647;
type FLOAT is digits 6 range -16#0.7FFF_FF8#E+32 .. 16#0.7FFF_FF8#E+32;

type LONG_FLOAT is digits 15 range -16#0.7FFF_FFFF_FFFF_FE#E+256 ..
        16#0.7FFF_FFFF_FFFF_FE0#E+256 ;
type DURATION is delta 0.0001 range -86400.0 .. 86400.0;
        -- DURATION'SMALL = 2#1.0#E-14 (that is, 6.103516E-5 sec)
...
end STANDARD;
```

5.9.4. Attributes of Type Duration

The type DURATION is defined with the following characteristics:

Attribute	Value
DURATION'DELTA	0.0001 sec
DURATION'SMALL	$6.103516E^{-5}$ sec
DURATION'FIRST	-86400.0 sec
DURATION'LAST	86400.0 sec

5.9.5. Values of Integer Attributes

Tartan Ada supports the predefined integer types INTEGER, SHORT_INTEGER and BYTE_INTEGER. The range bounds of these predefined types are:

Attribute	Value
INTEGER'FIRST	-2**31
INTEGER'LAST	2**31-1
SHORT_INTEGER'FIRST	-2**15
SHORT_INTEGER'LAST	2**15-1
BYTE_INTEGER'FIRST	-128
BYTE_INTEGER'LAST	127

The range bounds for subtypes declared in package TEXT_IO are:

Attribute	Value
COUNT'FIRST	0
COUNT'LAST	INTEGER'LAST - 1
POSITIVE_COUNT'FIRST	1
POSITIVE_COUNT'LAST	INTEGER'LAST - 1
FIELD'FIRST	0
FIELD'LAST	20

The range bounds for subtypes declared in packages DIRECT_IO are:

Attribute	Value
COUNT'FIRST	0
COUNT'LAST	INTEGER'LAST
POSITIVE_COUNT'FIRST	1
POSITIVE_COUNT'LAST	COUNT'LAST

5.9.6. Values of Floating-Point Attributes

Tartan Ada supports the predefined floating-point types FLOAT and LONG_FLOAT.

Attribute	Value for FLOAT
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON	16#0.1000_00#E-4 (approximately 9.53674E-07)
SMALL	16#0.8000_00#E-21 (approximately 2.58494E-26)
LARGE	16#0.FFFF_F8#E+21 (approximately 1.93428E+25)
SAFE_EMAX	126
SAFE_SMALL	16#0.2000_000#E-31 (approximately 5.87747E-39)
SAFE_LARGE	16#0.3FFF_FE0#E+32 (approximately 8.50706E+37)
FIRST	-16#0.7FFF_FFC#E+32 (approximately -1.70141E+38)
LAST	16#0.7FFF_FFC#E+32 (approximately 1.70141E+38)
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	126
MACHINE_EMIN	-126
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

Attribute	Value for LONG_FLOAT
DIGITS	15
MANTISSA	53
EMAX	204
EPSILON	16#0.4000_0000_0000_000#E-12 (approximately 8.8817841970013E-16)
SMALL	16#0.8000_0000_0000_000#E-51 (approximately 1.9446922743316E-62)
LARGE	16#0.FFFF_FFFF_FFFF_E00#E+51 (approximately 2.5711008708143E+61)
SAFE_EMAX	1022
SAFE_SMALL	16#0.2000_0000_0000_000#E-255 (approximately 1.1125369292536-308)
SAFE_LARGE	16#0.3FFF_FFFF_FFFF_F80#E+256 (approximately 4.4942328371557E+307)
FIRST	-16#0.7FFF_FFFF_FFFF_FE#E+256 (approximately -8.988465674312E+307)
LAST	16#0.7FFF_FFFF_FFFF_FE0#E+256 (approximately 8.9884656743115E+307)
MACHINE_RADIX	2
MACHINE_MANTISSA	51
MACHINE_EMAX	1022
MACHINE_EMIN	-1022
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE